TR-07601-14

TAOS ENVIRONMENTAL SIMULATION SERVICES

Version 1.0

Developer's Guide

30 January 1998

Prepared Under:

Contract No. DACA76-94-C-0021

for

U.S. Army Topographic Engineering Center

Attn: CETEC-TD-SM

7701 Telegraph Road

Alexandria, VA 22315-386

Technical POC:

Robert A. Reynolds

rareynolds@tasc.com

(781) 942-2000 x3117

Adminstrative POC:

Ronald M. Ponikvar

rmponikvar@tasc.com

(781) 942-2000 x2986

TABLE OF CONTENTS

List Of Figures

List Of Tables

1. OVERVIEW

The material in this Developer's Guide is intended to provide a software engineer with an overview of the TAOS source code. The TAOS software design is described first at the class library level and then at the class level; for documentation at the method/function level the header files and source code comments are the best resources. This guide is organized as follows. Section 2 describes the compile-time requirements for the system. Section 3 describes TAOS at the process level and includes an overview of the TAOS messaging system. The messaging system provides an interprocess communication (IPC) infrastructure for the system. Section 4 provides an overview of the TAOS source tree and its build infrastructure (i.e., make files). The TAOS server is designed to run with or without a GUI and as a result the system divides naturally into a server subsystem and a user interface subsystem. Section 5 documents the libraries and programs that comprise the server, while Section 6 similarly documents the user interface.

In this document the names of directories, files, libraries and programs are indicated in **bold** type. Class libraries in particular are referred to via the name of the library with a **lib** prefix and the **.a** suffix omitted; for example **libFoundation** is used to refer to the Foundation class library that exists in binary form in file **libFoundation.a** at directory **TAOS_HOME/src/shared/foundation/lib/IRIX5**. Note that for convenience the path to the TAOS root directory is denoted as a bolded **TAOS_HOME**. Note also that in this document any directory paths which are indicated as relative paths are relative to **TAOS_HOME** -- the directory where TAOS was installed. For example, **src/shared/Network/src** refers to directory **TAOS_HOME/src/shared/Network/src**.

2. COMPILE-TIME REQUIREMENTS

TAOS Version 1.0 was developed on an SGI platform (Indigo 2 class) running IRIX5.3. This document assumes that the execution environment for TAOS will be IRIX5.3. Note also that the execution environment must include patches to support C++ exception handling (required by the STOW RTI). The quickest way to detect if this support is present is to look for a file **/usr/lib/libCsup.so**.

Information on the required hardware environment is located in the User Guide, Section 2: "Installation and Run-time Preparation".

The TAOS system is written in a combination of C++, C, Tk/Tcl and Perl. The majority of the system is in C++ and compiles with the SGI CC compiler, Version 4.0 with exception handling patches 1260, 1599, 1600, and 1628, using the make infrastructure included in the distribution. See Section 3 for details of the make infrastructure.

The majority of the user interface is X/Motif-based and is written in C; these components compile under the SGI cc (V4.0) compiler using the included make infrastructure. Selected graphical interfaces and the Chart Utility are written in Tcl/Tk, Version 8.0 -- the required version of the Tcl interpreter and

support libraries are included in the TAOS distribution and will not conflict with any Tk/Tcl installation already on the host machine.

The FTP Receiver processes are written in Perl and here too the required version of the Perl interpreter and support libraries are included in the TAOS distribution and will not conflict with any Perl already installed on the host system.

3. PROCESS ARCHITECTURE AND MESSAGING SYSTEM

As described in the User's Guide, TAOS runs in three primary modes and most subsystems support secondary modes as well, with the result that rarely are all system processes active within the same exercise or at any given time within an exercise. Figure 3-1 is a data flow diagram of TAOS for an unlikely (but possible) scenario where the server is running in Live Mode (both Integrator and Distributor active) with all Receivers enabled and configured to run in Remote Mode, with the Distributor's Monitor and the GriddedWx Editor active, and with the Database Browser and the Chart Assistant utility running.

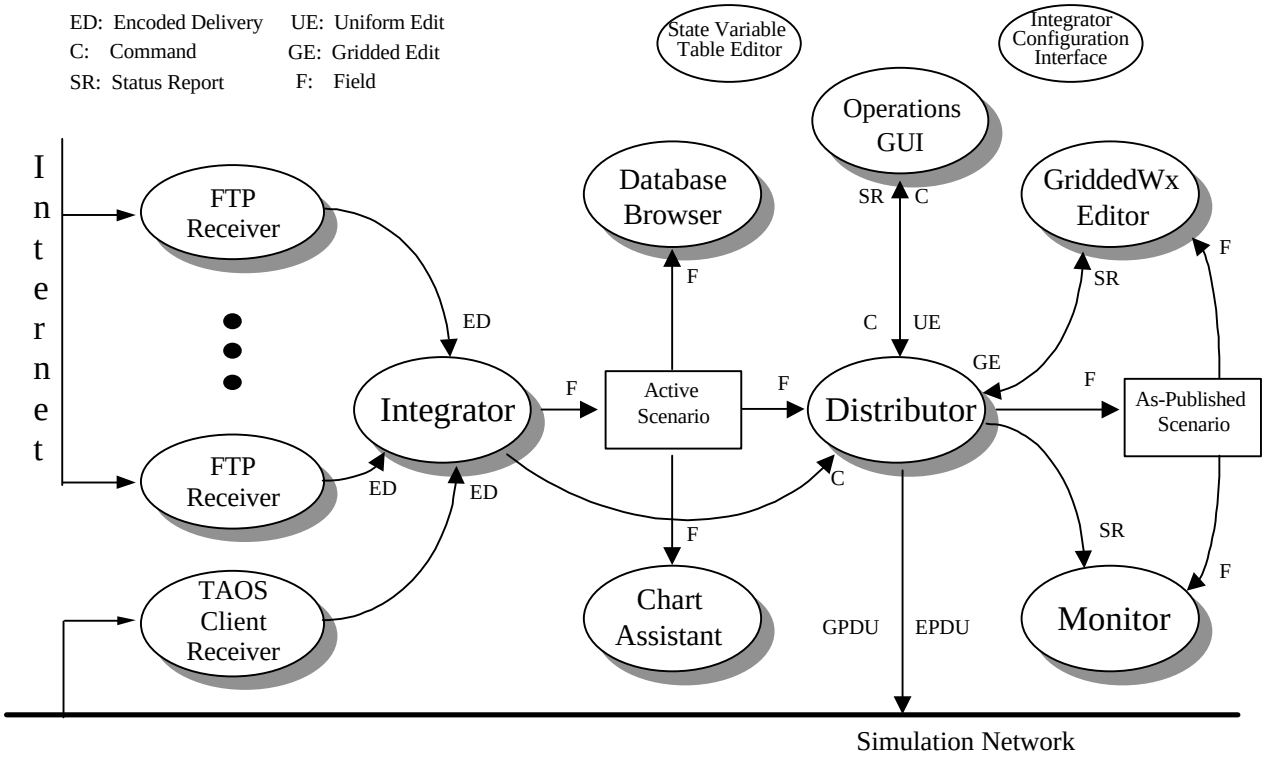


Figure 3-1 TAOS Processes and Data Flows

The ellipses in Figure 3-1 indicate TAOS processes, while the arrows indicate data flows between processes. Note that while the term *Operations GUI* is meant to refer to all graphical interfaces in the system from the end users' point of view, at the process level and from a developer's point of view the TAOS GUI is actually seven processes, with the "master" process given the name *OperationsGUI*.

These seven processes are identified in Table 3.1, which provides details of the full process architecture of TAOS.

Figure 3-1 describes system data flows for a fully initialized system running in "steady state". In this state all data exchanged between processes is either 1) message objects sent/received via the TAOS messaging system or 2) Field objects saved into or retrieved from the active and/or as-published scenarios. In Figure 3-1, each arrow head is annotated with a code that corresponds to the type of object sent over that link in the direction indicated by the arrowhead. Data flows consisting of Field objects are indicated with the code "F". All other codes denote objects of classes derived from *Message* which are sent and received using TAOS messaging system services (see **libMessaging** in Section 6.3). Table 3-2 defines the codes used in Figure 3-1 for these Message objects, the senders and receivers of these messages, and a brief description of their content.

Subsystem	Process	Program	Language	Location in Source Tree (from TAOS_HOME)
Integrator	Integrator	**EnvIntegrator**	C++	src/Integrator/Integrator
	TAOS Client Receiver	**TAOSClient**	C++	src/Integrator/receivers/DIS
	AWN Receiver COAMPS Receiver NOGAPS Receiver NORAPS Receiver NSSM Receiver OTIS Receiver WxRadar Receiver STWAVE Receiver SWAFS Receiver SWAPS Receiver Tide Receiver TOPS Receiver UKMeso Receiver WAM Receiver	**pollftp.prl**	Perl	src/Integrator/receivers/FTP
Distributor	Distributor	**EnvDistributor**	C++	src/Distributor
Operations GUI	Operations GUI	**UI**	C	src/OperationsGUI/XMotif/UI
	Database Browser	**DBBrowser**	C/C++	src/OperationsGUI/XMotif/DBBrowser
	GriddedWx Editor	**GriddedWxEditor**	C/C++	src/OperationsGUI/XMotif/GriddedWxEditor
	Monitor	**Monitor**	C/C++	src/OperationsGUI/XMotif/Monitor
	Chart Assistant	**taos_vis5dsh:** ChartAssistant.tcl	Tcl/Tk	src/OperationsGUI/TclTk/ChartAssistant
	Master State Variable Table Editor	**taos_vis5dsh :** MSVTEditor.tcl	Tcl/Tk	src/OperationsGUI/TclTk/MSVTEditor
	Integrator Configuration Interface	**taos_vis5dsh :** IntegConfig.tcl	Tcl/Tk	src/OperationsGUI/TclTk/IntegConfig
	Receiver Configuration Interface	**taos_vis5dsh :** RcvrConfig.tcl	Tcl/Tk	src/OperationsGUI/TclTk/RcvrConfig

Table 3-1 TAOS Subsystems, Processes and Programs

The TAOS messaging system models a message sender or recipient as an object of class *MessageBased*. Each MessageBased object has a unique address determined by the 1) TAOS process it

is instantiated in, 2) the host that this process is running on, and 3) an identifier for the object itself. To date all TAOS processes have always run on the same host; this level of addressing is for future use to facilitate scaleability of the server. The messaging system process and object identifiers are included in Table 2 for message senders and recipients. Object identifiers indicated in parentheses denote a message sender or recipient that is not implemented in C++ and is therefore not an actual MessageBased object. In particular, the FTP Receiver process (**pollftp.prl**) is implemented in Perl and sends messages using a direct Perl implementation of the UDP-based IPC used by **libMessaging**, and the Operations GUI accesses the services of **libMessaging** through a C API defined for the *PostOffice* class. In both of these cases, the sending/receiving entity uses an object identifier defined by the messaging system (see class *AddressBook*) but is not itself an object of (or derived from) class MessageBased.

As mentioned, Figure 3-1 illustrates data flows for steady-state server operation. During server configuration and initialization, the data flow is that which is illustrated in Figure 3-2. As suggested by Figure 3-2, a major function of the Operations GUI is to establish the configuration parameters for a server run that are read during initialization by the Integrator, Distributor, Receivers and Chart Assistant.

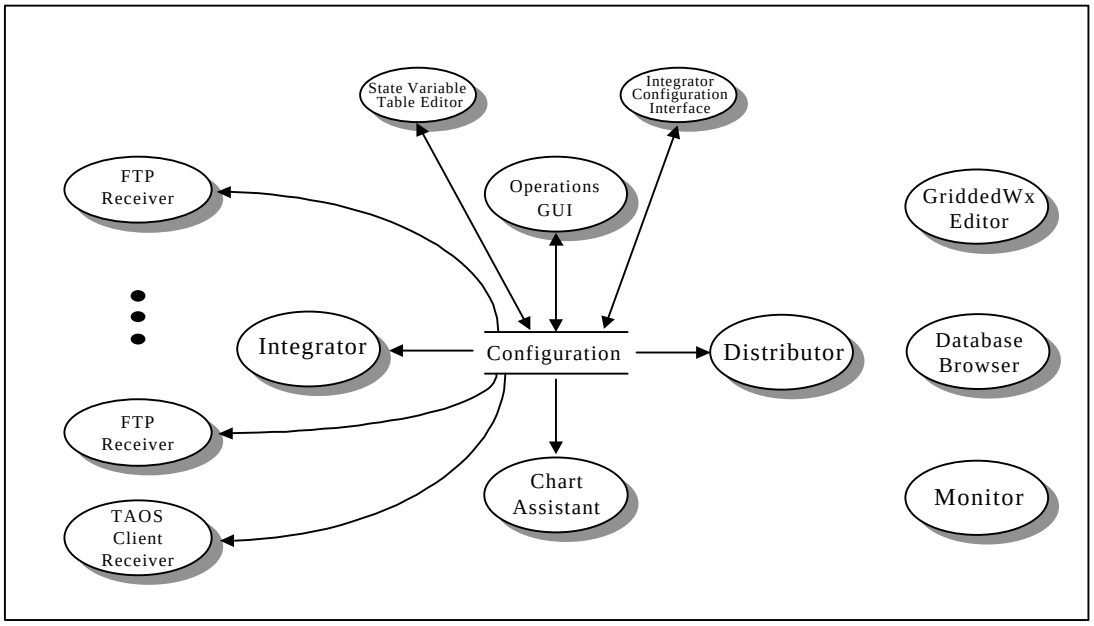


Figure 3-2 Pre-Exercise TAOS Data Flow

Another process-level view of TAOS is provided in Figure 3-3. In Figure 3-3 parent-child relationships between TAOS processes are indicated by arrows that are directed from parent to child. The root process is the OperationsGUI which the end user knows as **TAOS**, however it is possible to run the Integrator, Distributor and Database Browser in stand-alone mode by directly invoking their executables. In stand-alone mode, the Integrator, Distributor and Database Browser are driven by the TAOS configuration (directory) pointed at by environment variable **TAOS_CONFIG**.

Code	Message Class	Sent By		Received By		Message Content
		Process	Object	Process	Object	
ED	EncodedDelivery	FTP Receiver	(FTP Receiver)	Integrator	FTP Receiver	Pathnames to received files containing encoded environmental data.
		TAOS Client Receiver	TAOS Client Receiver	Integrator	TAOS Client Receiver	Pathnames of files containing Gridded Data PDU object images
C	Command	Integrator	Assimilation	Distributor	Publishing	EXECUTE subcommand to start publishing (Live Mode)
		Integrator	Assimilation	Distributor	Publishing	HOLD subcommand to suspend publishing during integration of new forecast data
		Operations GUI	(Playback Control)	Distributor	Publishing	CHECK_IN subcommand to request current playback-control-related status
		Operations GUI	(Playback Control)	Distributor	Publishing	PAUSE, STEP and PLAY subcommands and associated arguments
		Operations GUI	(UniformWx Editor)	Distributor	Publishing	CHECK_IN subcommand to request current edit-state of UniformWx Publication
		GriddedWx Editor	(GriddedWx Editor)	Distributor	Publishing	CHECK_IN subcommand to request current edit-state of GriddedWx Publication
		Monitor	(Monitor)	Distributor	Publishing	CHECK_IN subcommand to request current server status
		Distributor	Publishing	Operations GUI	(UI Master)	ENABLE_MONITOR subcommand, issued once virtual environment is initialized
		Distributor	Publishing	Operations GUI	(UI Master)	ENABLE_EDITORS subcommand, issued once virtual environment is initialized
		GriddedWx Editor	(GriddedWx Editor)	Distributor	GriddedWx Publication	CLEAR_EDB subcommand to clear all JointSAF/ModStealth GriddedWx Environmental DBs to allow reversion to UniformWx
SR	StatusReport	Distributor	Publishing	Operations GUI	(Playback Control)	Current server status including current Playback Control state
		Distributor	Publishing	Operations GUI	(UniformWx Editor)	Current server status including current Uniform Edit state
		Distributor	Publishing	Operations GUI	(NetSea Editor)	Current server status including current NetSea Edit state
		Distributor	Publishing	Monitor	(Monitor)	Current server status including network traffic statistics
		Distributor	Publishing	GriddedWx Editor	(GriddedWx Editor)	Current server status including current Gridded Edit state
UE	UniformEdit	Operations GUI	(UniformWx Editor)	Distributor	UniformWx Publication	Update to Uniform Edit specification
GE	GriddedEdit	GriddedWx Editor	(GriddedWx Editor)	Distributor	GriddedWx Publication	Update to Gridded Edit specification

Table 3-2 TAOS Messaging

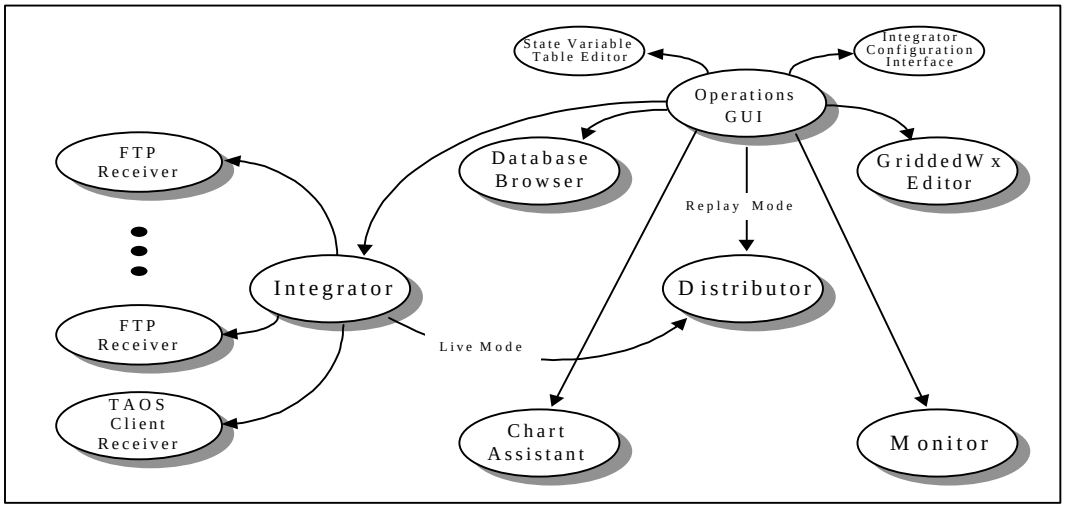
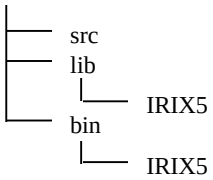


Figure 3-3 TAOS Process Group

4. SOURCE TREE OVERVIEW AND BUILD INFRASTRUCTURE

The TAOS source tree, rooted at **TAOS_HOME/src**, is shown in Figure 4-1. The leaf nodes of the tree shown in the figure correspond to buildable TAOS objects, either libraries (**lib** prefix) or programs (name in **bold**). Each node in the tree shown actually consists of a small tree structured as:



The make infrastructure is designed to detect the architecture of the machine on which the system is being compiled and to populate platform-specific directories with libraries and executables. To date only the IRIX5 platform has been supported. In any case the developer must create additional nodes under the **lib** and **bin** directories to support the platform-sensitive make procedures.

The TAOS make infrastructure is based on two master makefiles -- **project.mk** and **hierarchical.mk** -- located in directory **make**. **Makefile** are defined at various nodes (directories) of the source tree subject to the following simple rules:

1. If a node is not associated with a buildable library or program, then it is considered to be a parent node of a related group of libraries or programs, and must include a file named **Makefile** that defines the macro SRC_DIRS and includes **hierarchical.mk**, as in this example taken from **TAOS_HOME/src/shared**:

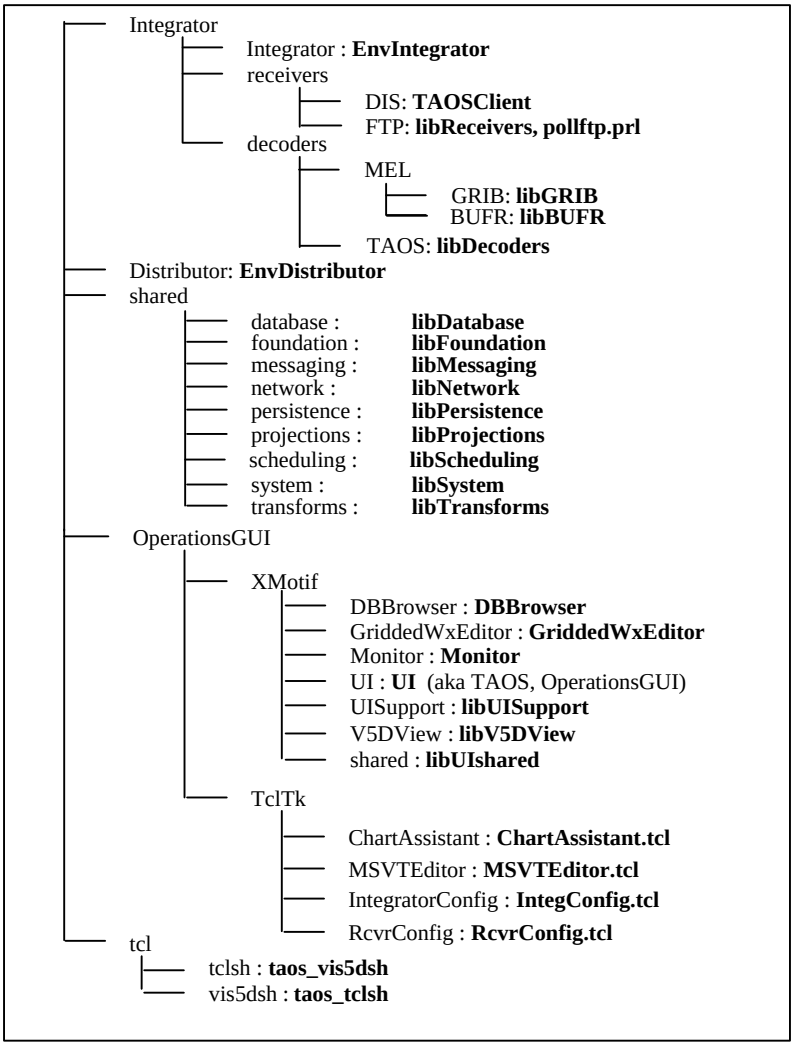


Figure 4-1 TAOS Source Tree

```
SRC_DIRS =      foundation/src  \
                system/src      \
                database/src    \
                messaging/src   \
                persistence/src \
                transforms/src  \
                scheduling/src  \
                network/src \
                projections/src

include hierarchical.mk
```

The directories included in the SRC_DIRS definition must either be buildable nodes or themselves be parent nodes.

2. If a node *is* associated with a buildable library or program, then it must include a **Makefile** that defines the macros found in **TAOS_HOME/make/Makefile.template**

and which includes **project.mk** in the last line of the file. For example, the **Makefile** for the foundation library is:

```
LIB_NAME = libFoundation.a
LIB_LOCATION = ../lib
LIB_SOURCE =    Boolean.cc \
                Error.cc \
                String.cc \
                ConfigFile.cc \
                Vector3D.cc \
                Table.cc \
                RunTimeEnv.cc \
                Time.cc

EXE_NAME =
EXE_LOCATION = ../bin
EXE_SOURCE =
EXE_LIBS =
INSTALL_LIBS = ${LIB_NAME}
INSTALL_FILES = *.hh

USER_CCFLAGS =
USER_LDFLAGS =
include project.mk
```

Only the macros relevant to building a given node need be given values; in the example there is no EXE_NAME, EXE_SOURCE or EXE_LIBS assigned because the node builds only a shared library, **libFoundation.a**.

The TAOS make infrastructure is known to work correctly with the ClearCase version of make, **clearmake**, which provides header file dependency checking, and with the GNU make (Version 3.74), **gmake**. However, the header file dependencies are not detected when using **gmake**. It is known that the system will *not* build using the SGI version of **make**, which is relatively primitive.

Due to the hierarchical design of the make infrastructure, the system can be fully built by issuing command **make** from the TAOS home directory. All libraries and executables will be built as make descends the source tree. Specific make targets are also supported:

- **make lib** -- build only libraries
- **make exe** -- build libraries *and* executables. Equivalent to **make** with no targets.
- **make clean** -- remove all **.o**, **.a**, executables, and emacs backup files.

In all cases, issuing a make command from a given directory results in a recursive descent of the source tree from that directory, affecting all nodes encountered in this descent.

5. SERVER

The TAOS system is organized as two subsystems: the *server* and the *Operations GUI*. The server subsystem comprises the Integrator, the Distributor and the Database and

implements the primary functions of the system: environmental data collection, transformation and integration, environmental data customization, distribution and related services, and environmental data persistence. The server is implemented largely in C++ and is documented in this section; Section 6 documents the Operations GUI, which implements all user-in-the-loop configuration, monitoring and control functions, including the environmental data visualization system.

5.1 Integrator Class Libraries and Programs

The Integrator *program* consists of a **main** function and the classes included in libraries **libIntegrator**, **libReceiver**, and **libDecoder**. The Integrator *subsystem* includes the Integrator program and the programs known as the *FTP Receiver* and the *TAOS Client Receiver*. These programs and libraries are documented in the subsections below.

5.1.1 **libIntegrator** (at: src/Integrator/Integrator)

libIntegrator includes classes *Assimilation*, *Receiving* and *Receiver*. The object model for these classes is represented in OMT notation[1] in Figure 5-1. Note that that we don't attempt to indicate individual class methods and data via the diagram; if the body of a class symbol in the figure is annotated with anything, it is the name of the library that includes that class.

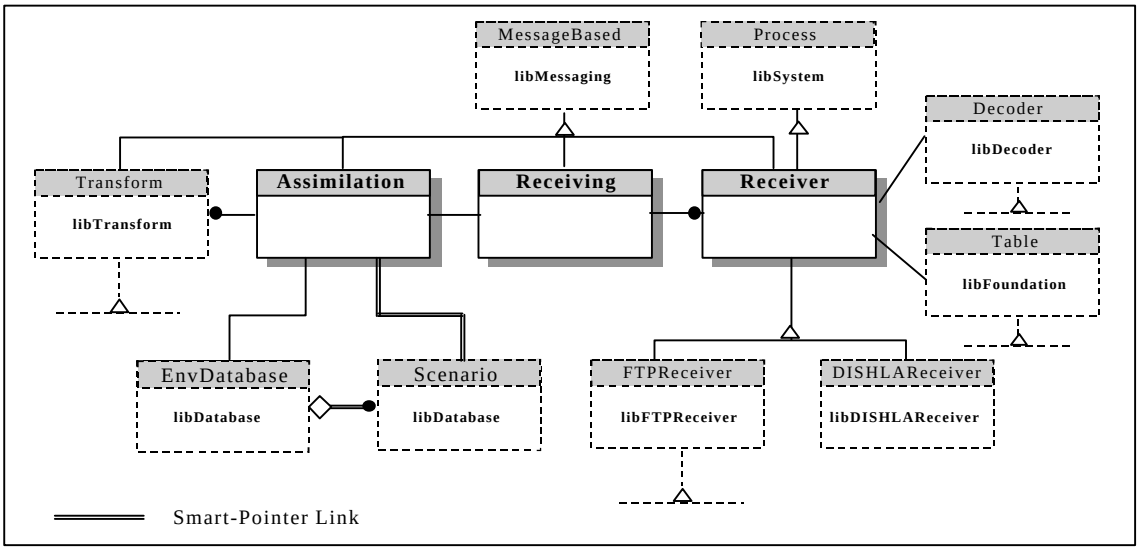


Figure 5-1 TAOS Object Model: libIntegrator

Classes Assimilation, Receiving and Receiver collaborate to perform the higher level functions of the Integrator. Classes Assimilation and Receiving are *single-instance* classes. The Integrator's **main** function instantiates the single object that "oversees" Integrator operations: the Assimilation object. Assimilation configures itself (reads **server.cfg**) and instantiates the Receiving object. During configuration Assimilation learns which Transforms the user has

[1] Rumbaugh, et. al., **Object-Oriented Modeling and Design**, Prentice Hall, 1991

enabled for the current server run and creates one instance of each enabled Transform. Assimilation also accesses the persistent Scenario object that the user has designated to be the active scenario for the server run; if this object doesn't exist, it is created. During configuration Receiving learns which Receivers the user has enabled and creates one instance of each. Each Receiver is then directed to configure itself by Receiving. Configuration errors result in shutdown.

All classes in **libIntegrator** are MessageBased and are therefore designed to communicate with each other primarily through exchange of TAOS message objects (see Section 5.3.5). Although the messaging system design anticipates that MessageBased objects will be run-time configurable with respect to the host/processor that they are to be executed on, Version 1.0 assigns all MessageBased objects to either the Integrator process or the Distributor process and all TAOS processes to the local host. It is assumed that the local host has a single processor. The assignments are made statically in file **AddressBook.cc** of **libMessaging**. In Version 1.0 Assimilation, Receiving and all Receiver objects are assigned to the Integrator process, and the messages exchanged by these classes therefore represent *intra*-process communication which is not shown in Figure 3-2; Figure 3-2 indicates only *inter*-process communication.

The intra-process messaging consists largely of sends/receives of *FieldDelivery* messages (see **libMessaging**) between the Receivers and Assimilation. Receiving's role is limited once the server is initialized, acting as a thin management layer between Assimilation and the Receivers. The Receiver objects act as sources of Field objects (delivered via a FieldDelivery message) and Assimilation acts as the sink. Receivers process a "stream" of input files, converting each to a Field object using a set of abstract, polymorphic environmental data pre-processing operations (methods) such as *decode*[2], *register*, *crop* and *preprocess*, the latter being a hook for an arbitrary data conditioning procedure.

Assimilation handles received Field objects without consideration of their exact type (see **libDatabase**). Each Field received is presented as a candidate input field to each of the enabled Transform objects. The Field is then stored in the active scenario via a call to *Scenario::add_field* followed by a call to *Scenario::save*. It is a policy that Transform objects make copies of input Fields that they are interested in, so that Assimilation is able to destroy the Field once it has been stored.

If when presented a candidate input field a Transform "fires" and consumes its input Fields to create its output Field(s), Assimilation either receives a Field object directly from the Transform or the Transform sends a FieldDelivery message to Assimilation which Assimilation will find when it next checks its "mailbox" (see **libMessaging**). Either way Assimilation ensures that the output Field(s) are in turn presented as candidate inputs to the active Transforms. Hence Assimilation behaves like a "Field pump" with respect to the set of active Transforms, ensuring that every Transform has an opportunity to act on every Field received directly from a Receiver or output from itself or another Transform.

[2] The polymorphic "decode" method is actually *Decoder::next_field*.

5.1.2 **libFTPReceiver** (at: src/Integrator/receivers/FTP)

libFTPReceiver includes the *FTPReceiver* base class which implements the Receiver functions specific to remote file access via the well-known Internet File Transfer Protocol. This functionality is entirely contained within method *FTPReceiver::execute*, which when invoked delegates essentially all of the responsibility for communication with the remote server to the Perl program (script) **pollftp.prl**. **pollftp.prl** is located in the **bin/IRIX5** directory associated with the **libFTPReceiver** node and is described in Section 5.1.4 below. **pollftp.prl** is actually executed in method *FTPReceiver::configure* but the process is designed to suspend attempts to connect with the remote FTP server until given a "go-ahead" signal (literally SIGUSR1); the *FTPReceiver::execute* method sends this signal.

The remaining classes in the **libFTPReceiver** library are each specific to an external model or data source:

Ocean Regime FTP Receiver Classes:	*OTISReceiver* *SWAFSReceiver* *TOPSReceiver*
Wave/Surf Regime FTP Receiver Classes:	*NSSMReceiver* *STWAVEReceiver* *SWAPSReceiver* *TideReceiver* *WAMReceiver*
Atmospheric Regime FTP Receiver Classes:	*AWNReceiver* *COAMPSReceiver* *NOGAPSReceiver* *NORAPSReceiver* *UKMesoReceiver* *WxRadar Receiver*

Many of these classes implement the minimum required specialized methods, inheriting all required functionality from the FTPReceiver and Receiver base classes.

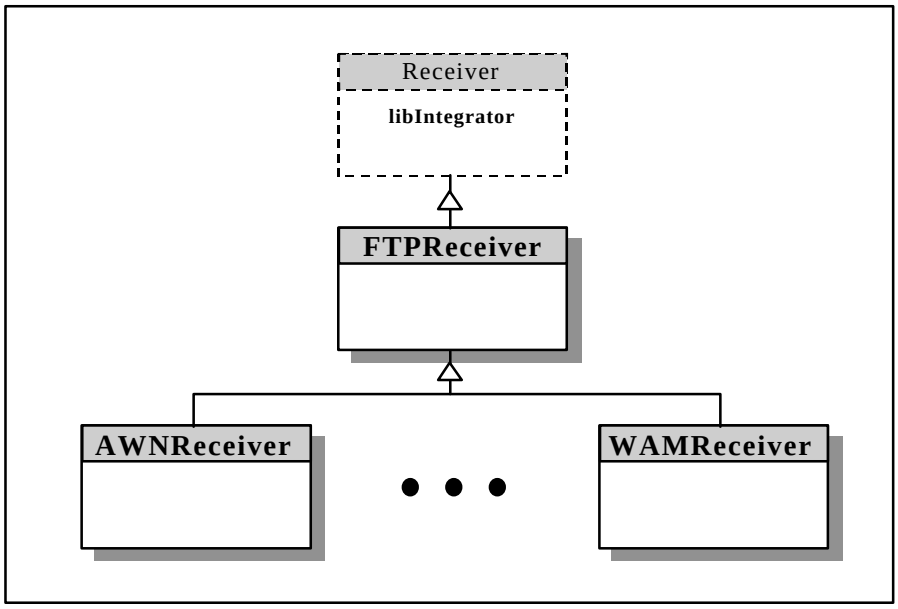


Figure 5-2 TAOS Object Model: libFTPReceiver

5.1.3 **libDecoder** (at: src/Integrator/decoders/TAOS)

A TAOS *decoder* is an object that knows how to extract Field objects from disk files. The model presumes that multiple Fields may be stored in the same file. Abstract base class *Decoder* defines the interface that a derived class must provide. Class Decoder's primary public methods are:

- *Error Decoder::bind_to_pathname(const String& encoded_file_pathname)* -- associates a file with a decoder.
- *Error Decoder::next_field(Field *& decoded_field_p)* -- successive calls update the argument until there are no fields left to decode, as indicated by a NULL argument update.
- *Error Decoder::unbind(void)* -- disassociates the decoder from the encoded file currently bound

The *next_field* method is pure virtual and must be defined for a specific encoding. Specific encoded file types supported by TAOS Version 1.0 are:

- *GRIB* -- the WMO GRIdded Binary format for gridded meteorological data
- *BUFR* -- the WMO Binary Universal FoRmat for (largely observational) meteorological data

- *WSIBMP* -- an extension of the MS Windows **.bmp** format used by WSI Corporation's Weather for Windows product for surface radar imagery.

- *Tide* -- not formally defined: a simple ASCII format used by the XTide program.

Figure 5-3 illustrates the **libDecoder** class hierarchy.

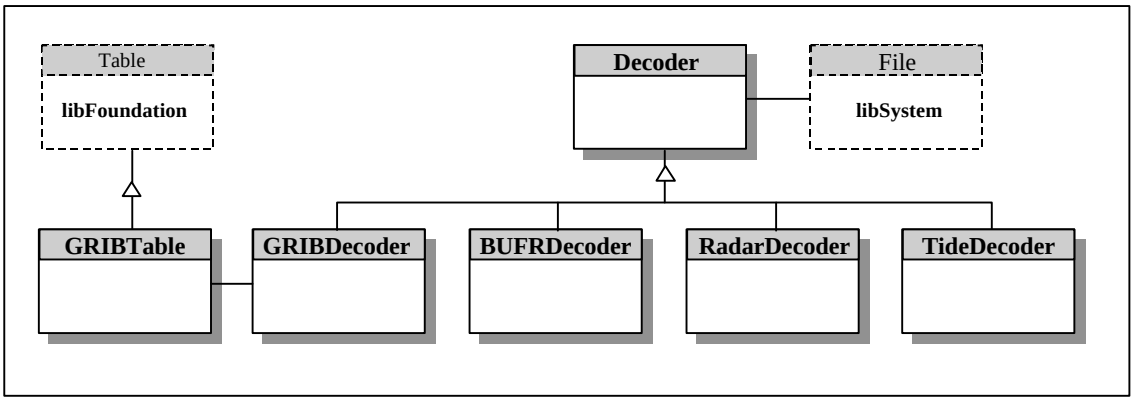


Figure 5-3 TAOS Object Model: libDecoder

5.1.4 **FTP Receiver** (at: src/Integrator/receivers/FTP/bin/IRIX5)

The FTP Receiver program is the Perl script **pollftp.prl**, located at **src/Integrator/receivers/FTP/bin/IRIX5**. This script requires Version 4.036 of Perl; this version is included in the TAOS distribution at **TAOS_HOME/perl/perl4.036**. Public domain Perl libraries used by **pollftp.prl** are included in the bin directory given above. **pollftp.prl** reads configuration data using the TAOS **.cfg** file format, and sends messages via the TAOS messaging system, but does not use TAOS C++ shared library code for either function, employing instead a functionally-equivalent implementation in Perl. The **pollftp.prl** script is well-commented.

Each FTPReceiver object that is running in Remote Mode spawns its own **pollftp.prl** process for communicating with its external model or data source. **pollftp.prl** defines command-line parameters for passing a Receiver id to the script so that the appropriate **.cfg** file is read by the program. For example, the UKMesoReceiver object spawns an instance of **pollftp.prl** and passes it the string "UKMeso", which **pollftp.prl** uses to read configuration file **UKMeso.cfg**.

5.1.5 **TAOS Client Receiver** (at: src/Integrator/receivers/DISHLA)

The TAOS Client Receiver communicates with a second, "upstream" instance of a TAOS server via either the DIS 2.0.3 or 2.0.4 protocols, or via the protocol defined by the STOW Federated Object Model derived from these DIS protocols. This Receiver *listens for* (DIS) or *subscribes to* (HLA/RTI) Gridded Data PDUs (objects), acting effectively as the simulation client to the upstream GriddedEnvPublication object (**libDistributor**). The Integrator Transform

Field3D (**libTransform**) is designed to reassemble 3D gridded fields[3] from 2D component fields. The *TAOS Client* program is the TAOS Server program running in "client mode". Client mode is not formally defined -- i.e., it is not a member of the enumeration *ServerMode* defined in **Types.hh**. TAOS Client mode is enabled in any system configuration where the TAOS Client Receiver, Field3D Transform and GriddedEnv Publication objects are all enabled. A TAOS instance running in client mode is designed to 1) receive and reassemble gridded environmental data from the designated TAOS Server for the simulation exercise, 2) populate a local TAOS database with field objects reconstituted from the network data, and 3) make this data available to the true client simulation application via a TCP/IP socket connection, which the GriddedEnvPublication class object manages when running in *private* (vs public) mode (see **libDistributor**).

The software components that comprise the TAOS Client Receiver are:

- **libDISHLAReceiver** -- consists of base class *DISHLAReceiver*, which implements the communication functions required to receive environmental data over a DIS simulation network or within an HLA Federation, and derived class *TCReceiver* which implements a few higher level functions, such as subscription to *gridded* environmental data and filtering of this data based on client-specified configuration parameters.

- **TCReceiver.cc** -- the **main** function for the TAOS Client Receiver *program.* The **main** function instantiates a single TCReceiver object and gives it control of the process via a call to *TCReceiver::execute*. The TCReceiver class implements the functions of a process proxy as well as those of the process that it proxies.

- **TCReceiver** -- executable resulting from linking **TCReceiver.cc** with **libDISHLAReceiver**.

Figure 5-4 provides a context for the classes included in **libDISHLAReceiver**.

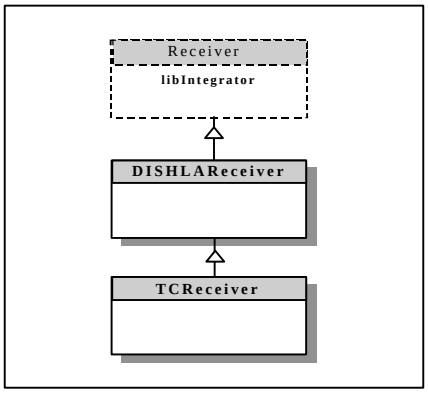


Figure 5-4 TAOS Object Model: libDISHLAReceiver

[3] Specifically, 3D rectangularly-gridded fields.

5.2 **Distributor Class Library** (at: src/Distributor)

Library **libDistributor** includes classes that provide services uniquely-required by the Distributor (program **EnvDistributor**). **EnvDistributor** is the executable built by linking **Distributor.cc**, containing the **main** function of the Distributor, with **libDistributor**. The **libDistributor** object model is illustrated in Figure 5-5.

Classes Publishing, Scheduler and all classes derived from Publication are single-instance classes. The **main** function for the Distributor instantiates a single Publishing object to act as a manager for the Distributor process. Publishing owns a set of Publication objects. The Publication base class defines public and protected interfaces for which derived Publications provide implementations.

The Distributor **main** function instantiates a Scheduler as part of its initial configuration procedure, then tells the Scheduler to configure itself. The Scheduler reads initial time and rate data from **server.cfg**, as specified via the GUI's Distributor Setup interface and then provides services to Publishing and the active Publications for scheduling regular environmental updates, polling of a messaging-system-provided mailboxes for communication with other TAOS processes, and other periodically-scheduled events.

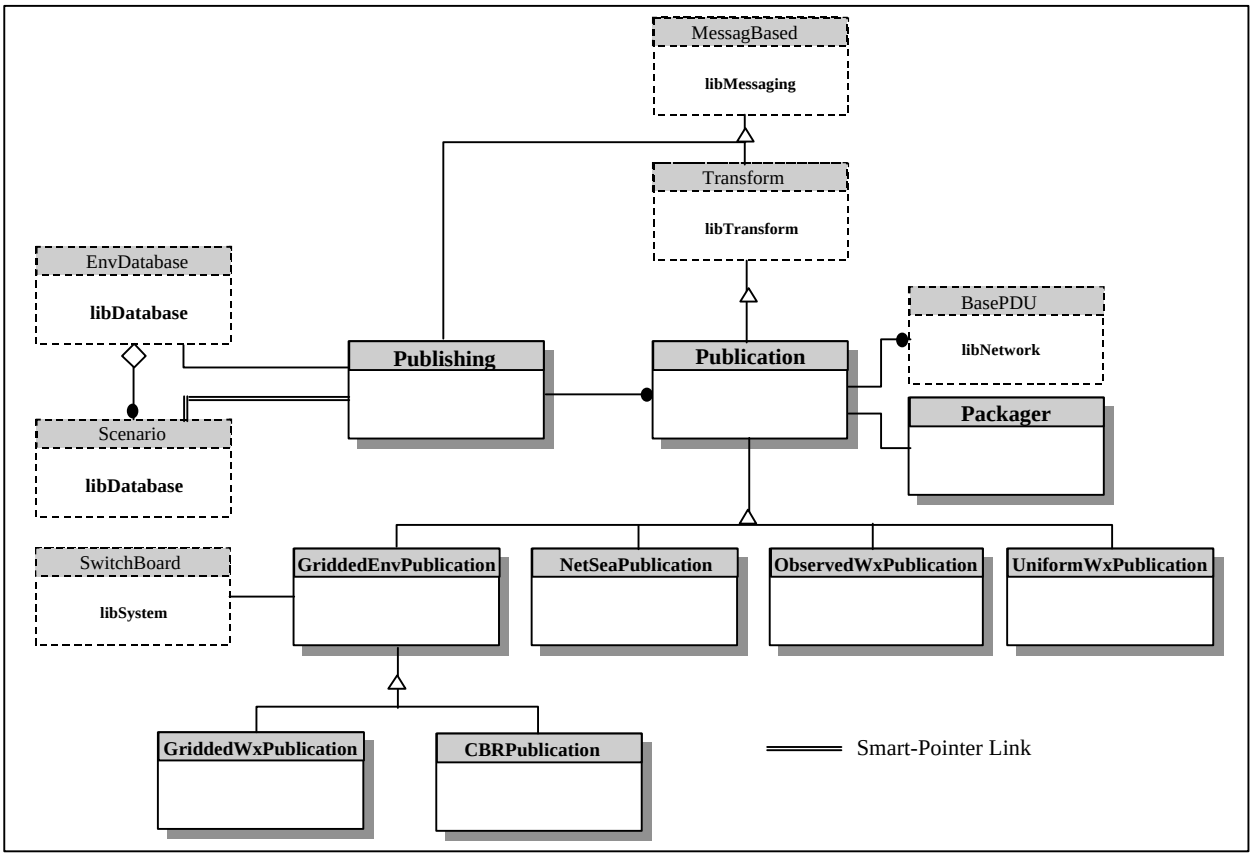


Figure 5-5 TAOS Object Model: libDistributor

As indicated in Figure 5-5, Publication objects are Transforms. Publications transform a set of Field samples[4] extracted by Publishing from the active Scenario object into network objects containing the environmental data required by client simulations and their effect models (JointSAF in particular). Each Publication owns a Packager object; the Packager provides methods for converting Field objects into network class objects derived from BasePDU (see **libNetwork**). PDU objects know how to send themselves over a DIS network or the RTI. At environmental update times derived Publication objects convert their input Fields into objects derived from BasePDU and tell these PDU objects to send themselves.

5.3 Shared Class Libraries (at: src/shared)

This section describes class libraries that provide common services to the Integrator, Distributor and Database subsystems. There are no executables associated with these nodes in the source tree.

5.3.1 libFoundation (at: src/shared/foundation)

libFoundation is the lowest level library in the TAOS system and includes container classes, a string class, and classes to handle errors and manage time. Much of the functionality provided by these classes is starting to become available in C++ Standard Template Library (STL) implementations, but at time of writing a STL for the IRIX5.3 platform is still not available. In many cases the TAOS foundation class implementations have been made compliant with the emerging STL standards. Figure 5-6 illustrates the classes included in **libFoundation**. Note that template classes are indicated as class names followed by **<T, … >**; for example, **Array<T>** denotes a templated container class parameterized on a single class **T**, while **Dictionary<K,T>** is a doubly-parameterized template class where **K** is the key class and **V** is the value class.

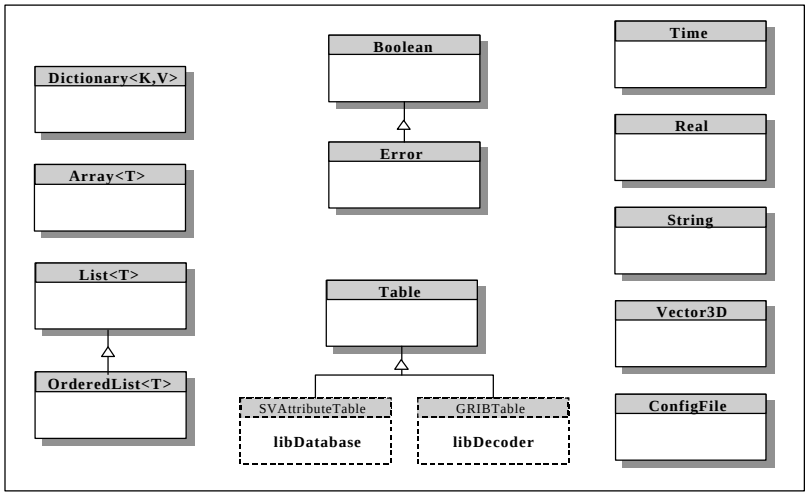


Figure 5-6 TAOS Object Model: libFoundation

[4] A Field sample is a 3D field extracted from a 4D field at a specific value of time

5.3.2 **libSystem** (at: src/shared/system)

libSystem is a low-level class library largely providing "object-orientation" for UNIX system functions. For example, the UNIX system calls for managing files and directories are encapsulated in methods defined on classes *File* and *Directory* respectively. Classes *UDPPort*, *TCPPort* and *SwitchBoard* encapsulate functions for creating and using datagram and stream sockets. Class *Console* provides facilities for building command-line oriented user interfaces and was used to build interim versions of the Publication Editors and Playback Control interface, however Version 1.0 employs graphical interfaces exclusively and class Console is not used. Class *Address* encapsulates methods for working with Internet addresses augmented with TAOS object identifiers and is used extensively by the messaging system (see **libMessaging**). The classes included in **libSystem** are indicated in Figure 5-7.

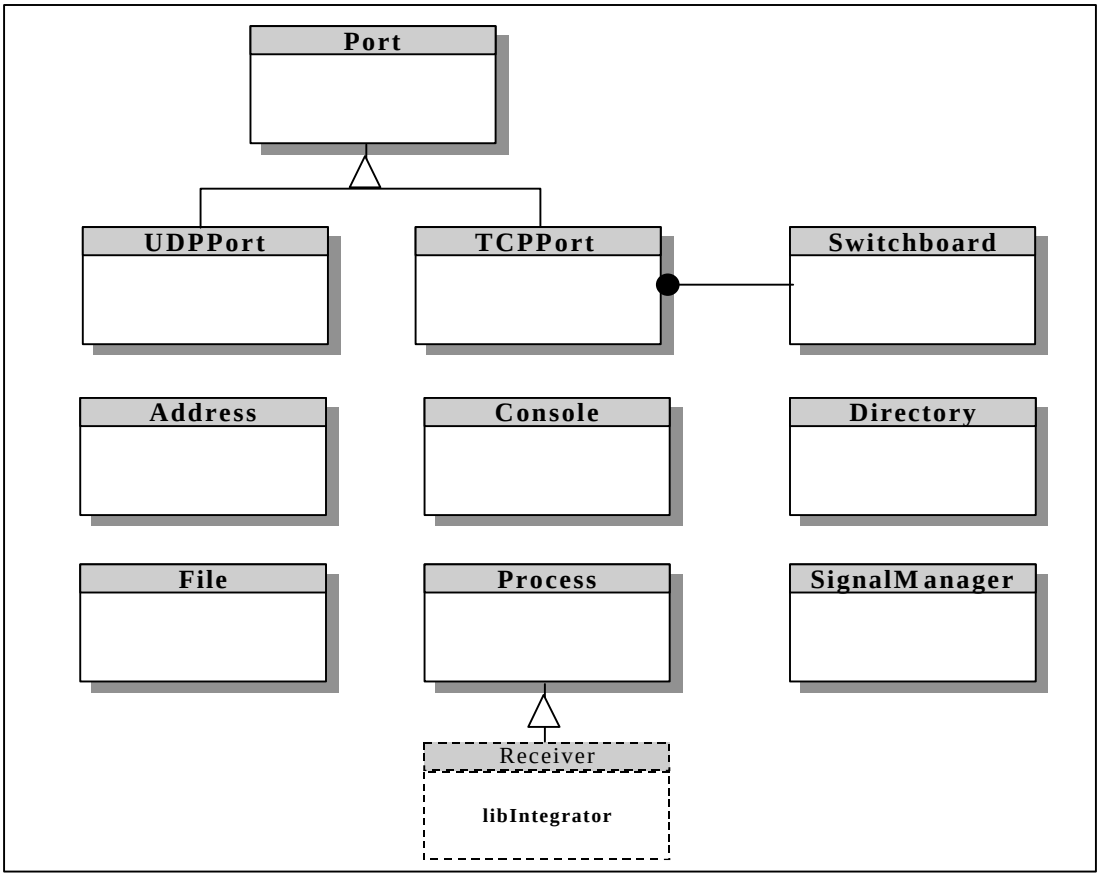


Figure 5-7 TAOS Object Model: libSystem

5.3.3 **libPersistence** (at: src/shared/persistence)

libPersistence provides persistent-object management services to the TAOS Database subsystem and to the messaging system; Figure 5-8 illustrates the object model for this library. The design of this class library was inspired by Shilling's article "How To Roll Your Own

Persistent Objects in C++" in the July 1994 issue of the Journal of Object-Oriented Programming (pp. 25–32).

The classes in **libPersistence** collaborate to allow client objects to create and manage *persistent object clusters***.** A persistent object cluster is a collection of persistent objects linked by smart pointers; a smart pointer is an object of the template class **SP<T>** that obeys "dumb" (ordinary) pointer syntax but overloads the **->** operator and effects automatic loading from disk when an object is referenced via the smart pointer for the first time. The primary cluster of persistent objects defined within TAOS is the environmental database itself; note however that **libPersistence** provides general persistent object management services and is not specific to the TAOS database – TAOS-database-specific classes are maintained separately in library **libDatabase**.

A persistent object cluster supports the following operations:

- *save* – synchronize disk image of object with memory image
- *restore* – synchronize memory image of object with disk image
- *free_disk* – release disk image, preserving memory image
- *free_memory* – release memory image, preserving disk image

When any of these methods are invoked on a persistent object, the object propagates the invocation to all persistent objects linked to that object, whether the link is via a dumb or smart pointer. Persistent objects maintain a "dirty bit" – a boolean variable indicating whether the (in-memory) object has been modified since last retrieved from disk. Methods **soil**, **clean** and **is_dirty** are used to manage the dirty bit, and component objects of a cluster which do not need to be saved (restored) are not.

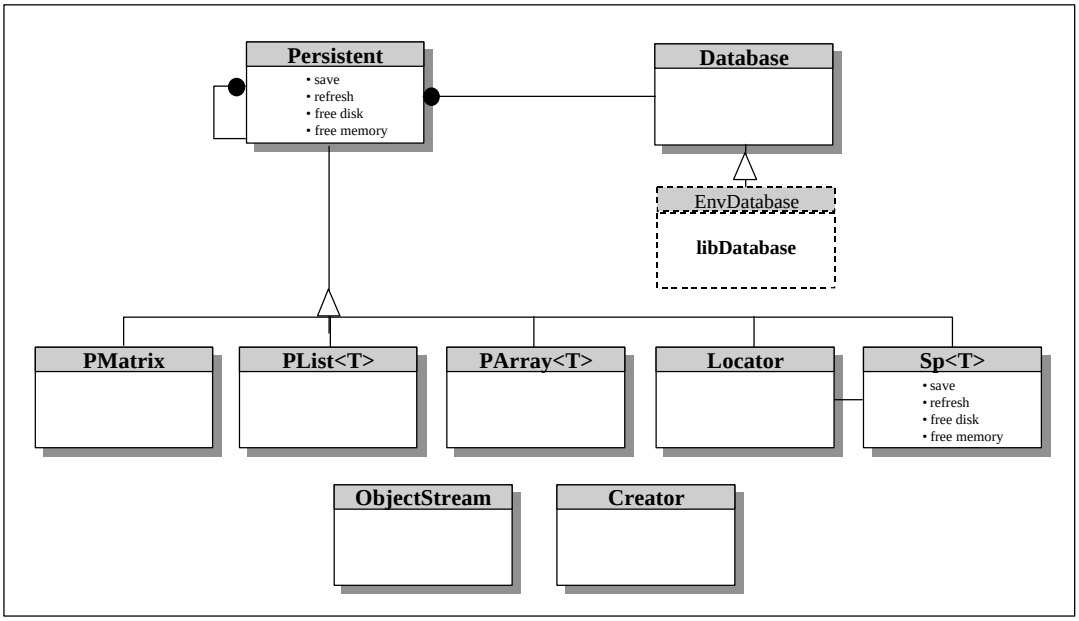


Figure 5-8 TAOS Object Model: libPersistence

Every Persistent object is linked to an associated *Database* object. In principle multiple Database objects may exist in the same process but in practice, and in TAOS Version 1.0 in particular, a single Database object is instantiated all Persistent objects are associated with this single instance. A Database object maintains a *root Locator* object which locates the persistent object cluster that this Database object is responsible for managing. A Locator is an opaque object (no public interface) whose internals are known only to the Database object (Database is a **friend** of Locator); a Locator locates the image of an object on disk. Every smart pointer embeds a Locator object which it requests from its Database when the object the smart pointer references is saved for the first time.

Classes *PList<T>* is a persistent-container class of general utility in constructing object clusters. The template class parameter must be a class derived from Persistent. Class *PMatrix* was used in an early implementation of **libDatabase**, but its functionality is now provided more efficiently by class PArray<float>; the class parameter for PArray<T> is designed to be a built-in atomic data type such as **float** or **char**. Class *Creator* encapsulates a table of pointers to "creator" functions, keyed on an enumerated set of class ids. Creator functions are static public member functions that each Persistent class must provide – these functions are called by a Database object to restore an object from disk based on the class id stored as the first field in the object's disk image. Class ObjectStream provides numerous operators for inserting and extracting objects into a serialized byte stream so that these objects can be stored on disk or transmitted over a network.

5.3.4 **libDatabase** (at: src/shared/database)

The object model for the classes included in **libDatabase** is indicated in Figure 5-9. These classes support management of the persistent object cluster which is the TAOS environmental database, and rely heavily on the services provided by **libPersistence**.

At the heart of **libDatabase** is the *Field* class sub-hierarchy, consisting of *Field*, *UniformField*, *WaveHeight* and *GField* (the latter is short for Gridded Field). Objects of specialized Field classes are at the leaves of the persistent object cluster. The *Scenario* class aggregates a collection of Field objects, and the *EnvDatabase* class aggregates Scenario objects. TAOS instantiates a single EnvDatabase object in each process that requires access to the active database for an exercise. This object can be queried for the Scenario objects it contains, and Scenario objects can be queried for the Field objects they contain.

A Field object is associated with a Measure object, which establishes the physical quantity represented by the field and the systems of units used to express this quantity; a Measure objects' primary responsibility is managing unit conversions for a particular physical quantity such as Temperature, Velocity or Pressure. TAOS Version 1.0 includes specialized Measure classes for these physical quantities:

- Temperature
- Pressure
- Distance
- Velocity
- Mixing Ratio
- Accumulated Precipitation
- Instantaneous Precipitation Rate
- Density
- Direction
- Wx Radar Reflectivity
- Frequency
- Duration
- Fraction / Percentage

All derived Measure classes are implemented in the **Measure.hh** and **Measure.cc** source files.

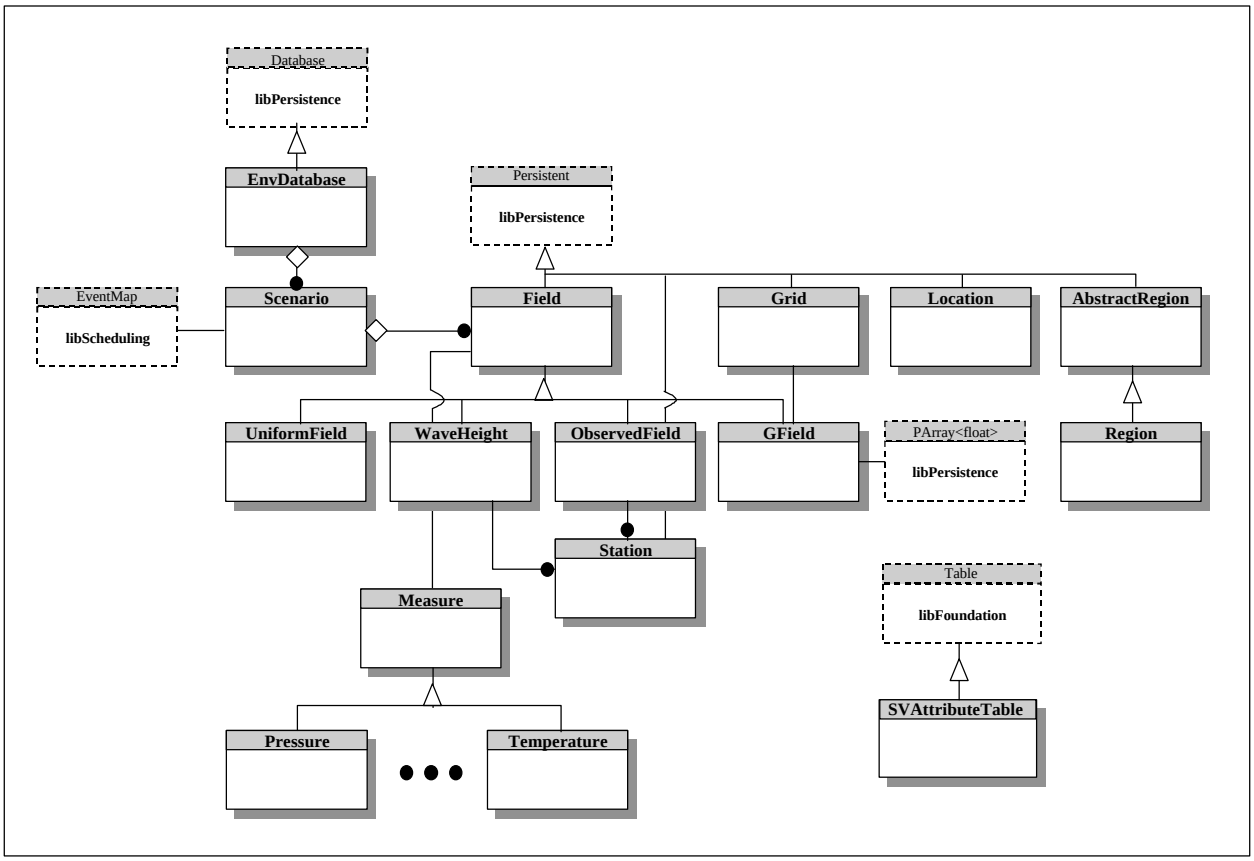


Figure 5-9 TAOS Object Model: libDatabase

5.3.5 **libMessaging** (at: src/shared/messaging)

The classes included in **libMessaging** are illustrated in the OMT diagram in Figure 5-10. All services associated with the TAOS messaging system are provided via this library.

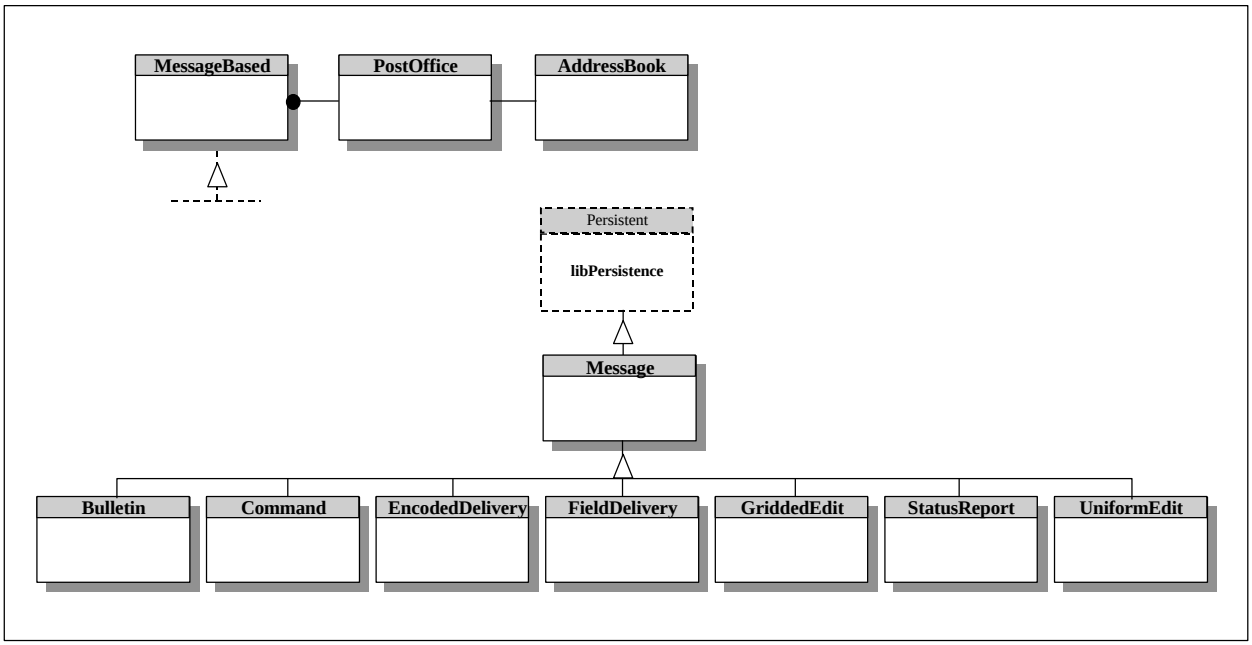


Figure 5-10 TAOS Object Model: libMessaging

As indicated in Figure 5-10, the classes that define **libMessaging** fall into two disjoint sets. Classes *MessageBased*, *PostOffice* and *AddressBook* provide the infrastructure for the dynamic components of the messaging system. The PostOffice is at the heart of the system and is a single-instance class (a single PostOffice object per process is allowed). A PostOffice object provides message delivery services for multiple MessageBased objects instantiated in its process. The PostOffice owns an AddressBook that contains addressing information for all MessageBased objects in all TAOS processes; MessageBased objects can access the AddressBook via an accessor provided by the PostOffice class. A PostOffice object provides services for registering a "mailbox" for each MessageBased object that it serves. A MessageBased object may poll its mailbox or ask the PostOffice to notify it of incoming messages via the protected callback function *MessageBased::notify*.

The classes derived from *Message* represent actual message objects sent and received by MessageBased objects, via the PostOffice. Class *Bulletin* is obsolete; it was used to encapsulate and transmit observational data before the advent of the ObservedField class (see **libDatabase**). Class *Command* defines a number of command-subtypes and supports arguments for these subtypes. Commands *GriddedEdit*, *StatusReport* and *UniformEdit* are used to exchange commands and status with the Publication Editors. See Table 3.2 for a summary of which message types are sent or received by which MessageBased objects.

5.3.6 **libTransforms** (at: src/shared/transforms)

The classes included in **libTransform** are illustrated in Figure 5-11.

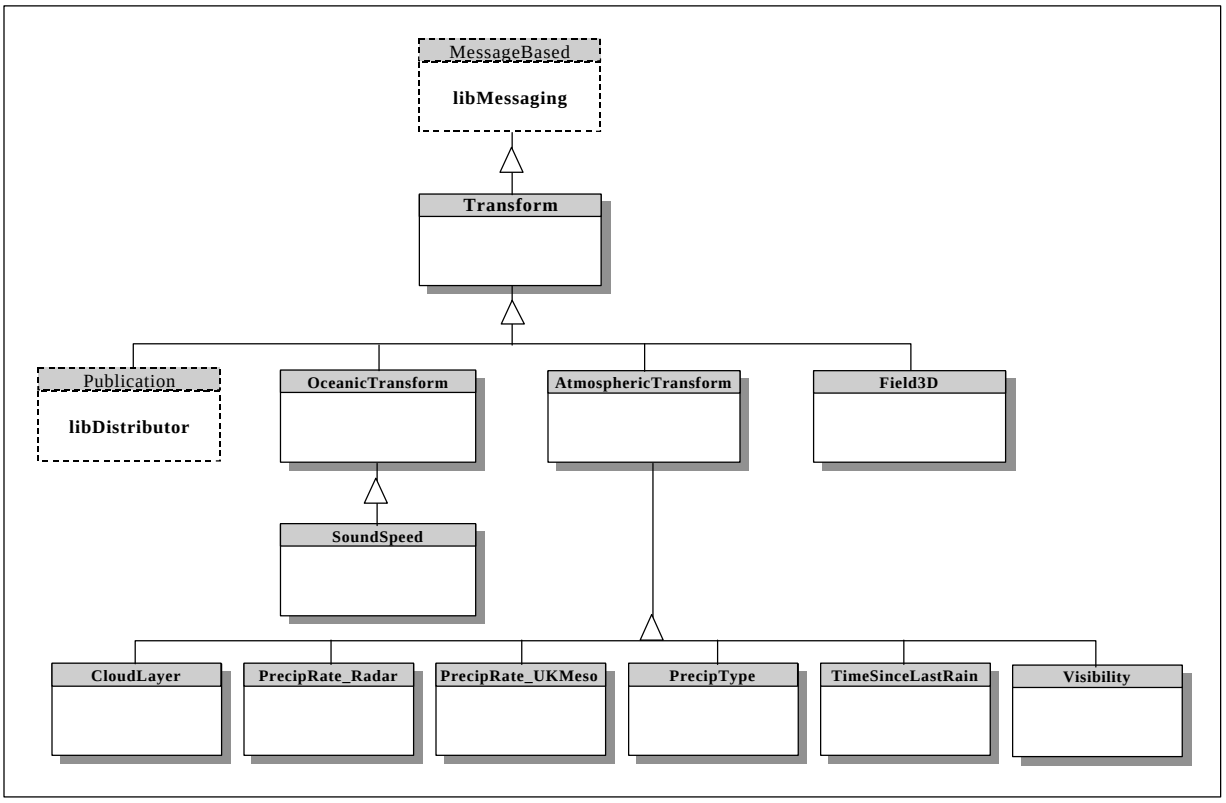


Figure 5-11 TAOS Object Model: libTransforms

A *Transform* is an object that accepts Field object inputs and which creates one or more output Fields when all of the required inputs have been received. A Transform is designed to accept one Field object at a time – either via method *Transform::input* or via receipt of a *FieldDelivery* message (Transforms are MessageBased objects). A Transform object treats a Field presented to it as a *candidate* input Field; if the Field is determined to be an actual input the Transform creates a private copy of the Field which it maintains until all inputs have been received.

Base classes *AtmosphericTransform* and *OceanicTransform* anticipate that Transforms within a specific environmental regime may benefit from a common set of base class methods. Class *Field3D* is not specific to a particular environmental regime. See the User's Guide for more information on the specifics of the specialized Transform classes at the leaves of the hierarchy in Figure 5-11.

5.3.7 **libNetwork** (at: src/shared/network)

libNetwork defines a hierarchy of environmental Protocol Data Units (PDUs) as indicated in Figure 5-12. The term "PDU" is used to refer to either a true DIS PDU or to an object published or subscribed to via the HLA's RTI.

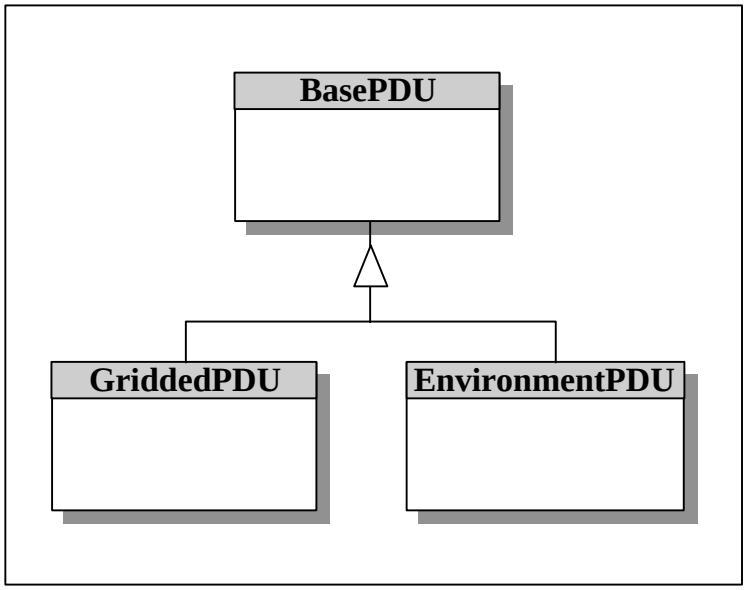


Figure 5-12 TAOS Object Model: libNetwork

Class ***BasePDU*** encapsulates attributes common to all environmental network objects. It also provides a key service method: *BasePDU::send_DSI* for sending the PDU to the Defense Simulation Internet (a generic term for the network layers of either a DIS- or HLA-based exercise). In addition to encapsulating the data for a specific environmental object, the BasePDU class is responsible for interfacing with the JointSAF **libpduapi** and **libpduproc** libraries; the latter provide simulation-architecture-independent network services that allow the same application to interoperate with DIS and HLA exercises.

5.3.8 **libScheduler (**at: src/shared/scheduling**)**

libScheduler is a small class library containing the classes indicated in Figure 5-13. Class *Scheduler* is at the heart of the library, providing key time-based scheduling services to the Distributor and supporting scheduling in real-time, simulation-time or historical-time. This class is based on the JointSAF **libsched** and **libtime** libraries. Class Scheduler is a single-instance class. The Scheduler object retrieves an *EventMap* object from the active Scenario object during configuration to use in determining the relationship between simulation time and historical time when this relationship is not linear and/or simply described. However, the EventMap class was developed to support a more complex mapping between historical time and simulation time than is currently supported by the Operations GUI's Playback Control Interface; class EventMap has been preserved within the design however in anticipation of future extensions to the Playback

Control subsystem. Class *Alarm* is a simple wrapper around the UNIX **alarm** system call and has been made largely obsolete by functions provided the Scheduler class.

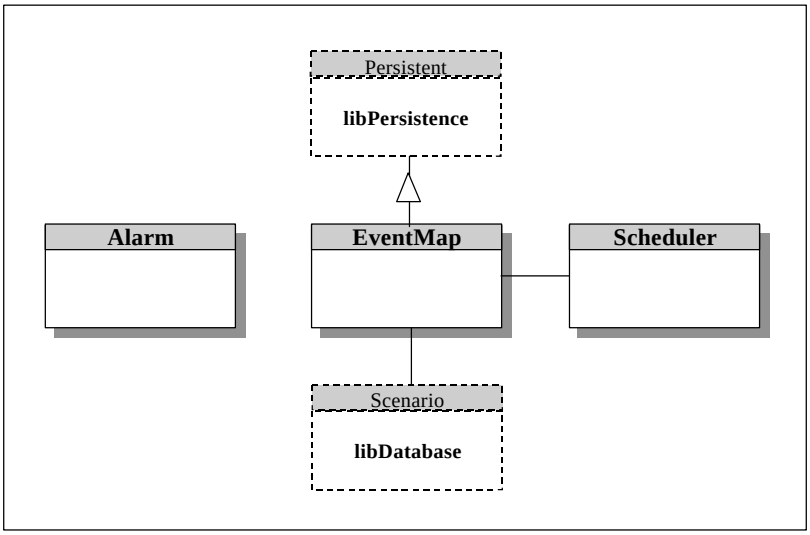


Figure 5-13 TAOS Object Model: libScheduler

5.3.9 **libProjections** (at: src/shared/projections)

libProjections is not a true class library but is instead a pair of functions for converting between Geodetic coordinates and coordinates defined on various standard projections supported by the GRIB standard (e.g., Lambert Conformal, Polar Stereographic). The C functions bundled into **libProjections** are based on Fortran code originally provided by NRL Monterey.

6. OPERATIONS GUI

The Operations GUI comprises all Graphical User Interfaces used for monitoring and control of the TAOS server. While the user sees these interfaces as a seamless whole, employing common look-and-feel, color schemes and fonts, they are actually implemented as a mix of 1) X/Motif programs written in C with interfaces built using the UIM/X GUI-builder and 2) Tcl/Tk programs written in Tcl with interfaces built at run-time directly from the Tcl scripts. The majority of the Operations GUI, including the TAOS Main Interface, is implemented in C using X/Motif and UIM/X. Section 6.1 documents the X/Motif programs. Section 6.2 documents the Tcl/Tk programs.

6.1 X/Motif GUI Components

The X/Motif-based GUI programs consist of the Operations GUI (the program which manages the TAOS main interface), the Database Browser, the Distributor Monitor and the Gridded Weather Editor. All are implemented in C using an object-based programming style. In

many cases these programs use services provided by the C++ libraries documented in Section 5; C++ classes whose services are required by X/Motif GUI programs define a C API that allow these services to be invoked from a C program.

The object-based design of the X/Motif components closely reflects the "object structure" of the interfaces as seen by the end user. In the sections which follow, the X/Motif components are described in terms of a hierarchy of these objects:

- *Program* – the X/Motif-based GUI components include the four distinct programs named above.
- *Interface* – also referred to as a *Dialog*, an interface is a top level window managed by the window manager. Each program creates and manages multiple interfaces.
- *Block* – a group of related widgets typically enclosed in a frame with a raised border with a label that refers to the group. Each interface creates and manages multiple blocks. Some blocks are used in multiple interfaces; management modules for these blocks are located in **src/OperationsGUI/XMotif/shared**.

In the sections below, the object structure of the X/Motif programs are presented in terms of tables which identify the interface/block hierarchy for the program and the C modules which manage objects in the hierarchy. C modules are identified by their module name: the source code that implements a module is located in files **<module-name>.h** and **<module-name>.c**; e.g., a module named ErrorDialog is implemented in source files **ErrorDialog.h** and **ErrorDialog.c**.

6.1.1 Operations GUI (at: src/OperationsGUI/XMotif/UI)

The Operations GUI program manages the TAOS main interface, the Distributor Configuration interface, the Publication configuration interfaces, the Playback Control interface, the UniformWx Editor and the NetSea Editor. The object structure of this program is shown in Table 6-1. The **main** function for this program is in source file **OperationsGUI.c**.

6.1.2 Database Browser (at: src/OperationsGUI/XMotif/DBBrowser)

The Database Browser program manages the Database Browser interface and the Event Definition interface. The object structure of this program is shown in Table 6-2. The **main** function for this program is written in C++ and is in file **DatabaseBrowser.cc**.

Interface	Block	Management Modules
Main	all	MainDialog DistributorStatusDisplay IntegratorStatusDisplay ReceiverStatusDisplay ServerStatusDisplay OpGuiReceivers StatusColors
Distributor Configuration	all	DistributorSetupDefinitions DistributorSetupDialog
	Exercise Parameters	ExerciseParmsBlock
	Times (UTC)	ExerciseTimesBlock
	Simulation Rate	SimulationRateBlock TimePeriodInputForm
	At Historical End Time	HistoricalEndBlock
	Publications	PublicationsBlock
GriddedWx Publication Configuration	all	ConfigGriddedDialog
	Load Leveling Window	LoadLevelingBlock
NetSea Publication Configuration	all	ConfigNetSeaDialog
UniformWx Publication Configuration	all	ConfigUniformDialog
Playback Control	all	PlaybackControlDialog
UniformWx Editor / NetSea Editor	all	UninetEditorDialog UninetEditBlock UninetSVeditor

Table 6-1 Operations GUI Object Structure

Interface	Block	Management Modules
Database Browser	all	DatabaseBrowseDialog
	Events	EventEditorBlock
	Scenario Bounds	ScenarioBoundsBlock
	State Variables	StateVariablesShowBlock
	(time advancement control)	V5dControlBlock
Event Definition	all	EventDefineDialog EventEditDialog

Table 6-2 Database Browser Object Structure

6.1.3 Distributor Monitor (at: src/OperationsGUI/XMotif/Monitor)

The Distributor Monitor program manages the Monitor interface and the Network Traffic interface. The object structure of this program is shown in Table 6-3. The **main** function for this program is in file **Monitor.c**.

Interface	Block	Management Modules
Monitor	all	MonitorDialog
Network Traffic	all	NetworkTrafficDialog

Table 6-3 Distributor Monitor Object Structure

6.1.4 GriddedWx Editor (at: src/OperationsGUI/XMotif/GriddedWxEditor)

The GriddedWx Editor program manages the GriddedWx Editor interface and the Insert Geometry interface. The object structure of this program is shown in Table 6-4. The **main** function for this program is in file **GriddedWxEditor.c**.

Interface	Block	Management Modules
GriddedWx Editor	all	GriddedWxEditorDialog
	Insert Editor	InsertEditBlock InsertsSet
Insert Geometry	all	GeometryDialog

Table 6-4 GriddedWx Editor Object Structure

6.1.5 Shared Libraries

Three shared libraries are used extensively by the X/Motif GUI programs. **libUIshared** contains C modules for managing widget blocks which appear on multiple interfaces. **libV5DView** is a C++ class library which provides an object-oriented interface to the Vis5D API for OpenGL-based environmental data visualization as well as OpenGL-based drawing algorithms for use in the GriddedWx Editor. **libUISupport** is an older library containing additional GUI support functions.

libUIshared (at: src/OperationsGUI/XMotif/shared)

libUIshared includes C modules used by multiple programs. These modules can be further classified on the basis of the services they provide:

- *Custom Widgets* – specialized compound widgets
- *Common Blocks* – blocks used in multiple interfaces
- *Utility Dialogs* – e.g., error and confirmation dialogs
- *Utility Modules* – utilities with no graphical counterpart, e.g. containers

Table 6-5 lists the modules in **libUIshared** by "service category".

Service Category	Modules
Custom Widget	BooleanMenuInput ControlScale EventSetDisplay IntegerInput OptionMenuInput RealInput ScaleInput StringInput
Common Blocks	ClockBlock DataViewBlock IsosurfaceForm ScalarHorizSliceForm ScalarVertSliceForm ScenarioSelectBlock StateVariableSelectBlock TimeEntryForm VectorHorizSliceForm VectorVertSliceForm VolumeForm
Utility Dialogs	ConfirmationDialog DirectorySelectionDialog ErrorDialog MessageDialog
Utility Modules	BooleanConfigParm ConfigParameterSet ConfigurationParameter Dialog EventSet FloatEditor IntegerConfigParm OpGUIpostOffice QStringConfigParm RealConfigParm RTConfig ScenarioView SelectList StateVariableEditor StateVarHolderList StateVariableHolder StateVariableSet StringConfigParm TimeConfigParm

Table 6-5 libUIshared Modules by Service Category

libV5DView (at: src/OperationsGUI/XMotif/V5DView)

The **libV5DView** class library provides an extensible interface for embedding interactive visualizations of large 5-D gridded data sets into an X /Motif application. This library provides an additional, object-oriented layer of abstraction above the University of Wisconsin Space

Science and Engineering Center Vis5D application/API that it is based on. **libV5DView** encapsulates most of the details of configuring an X/Motif-based graphics display for OpenGL rendering. Figure 6-1 illustrates the full set of system and application-specific libraries, and the Application Programming Interfaces (APIs) involved, in an application that combines X/Motif with OpenGL (on an IRIX5.x platform) via the **libV5DView** library. The object model for the classes in the **libV5DView** library is shown in Figure 6-2.

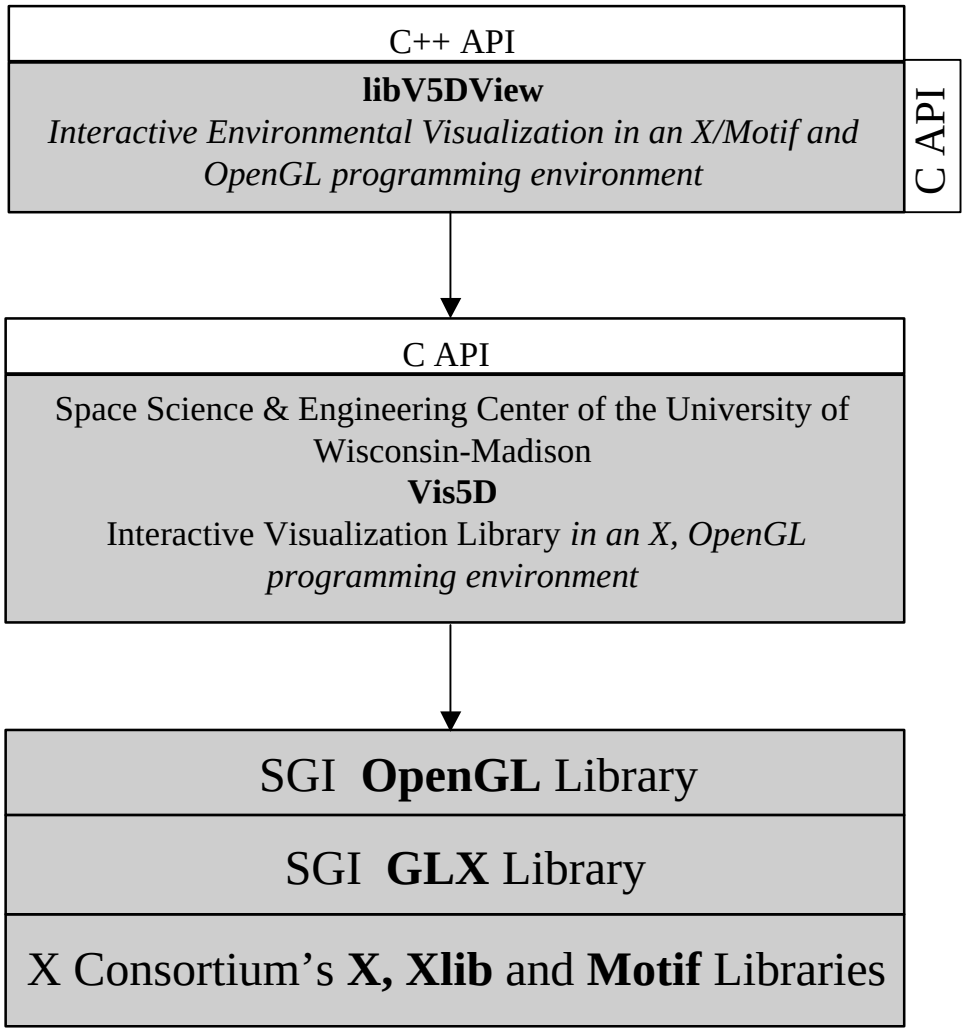


Figure 6-1 libV5DView Context and APIs

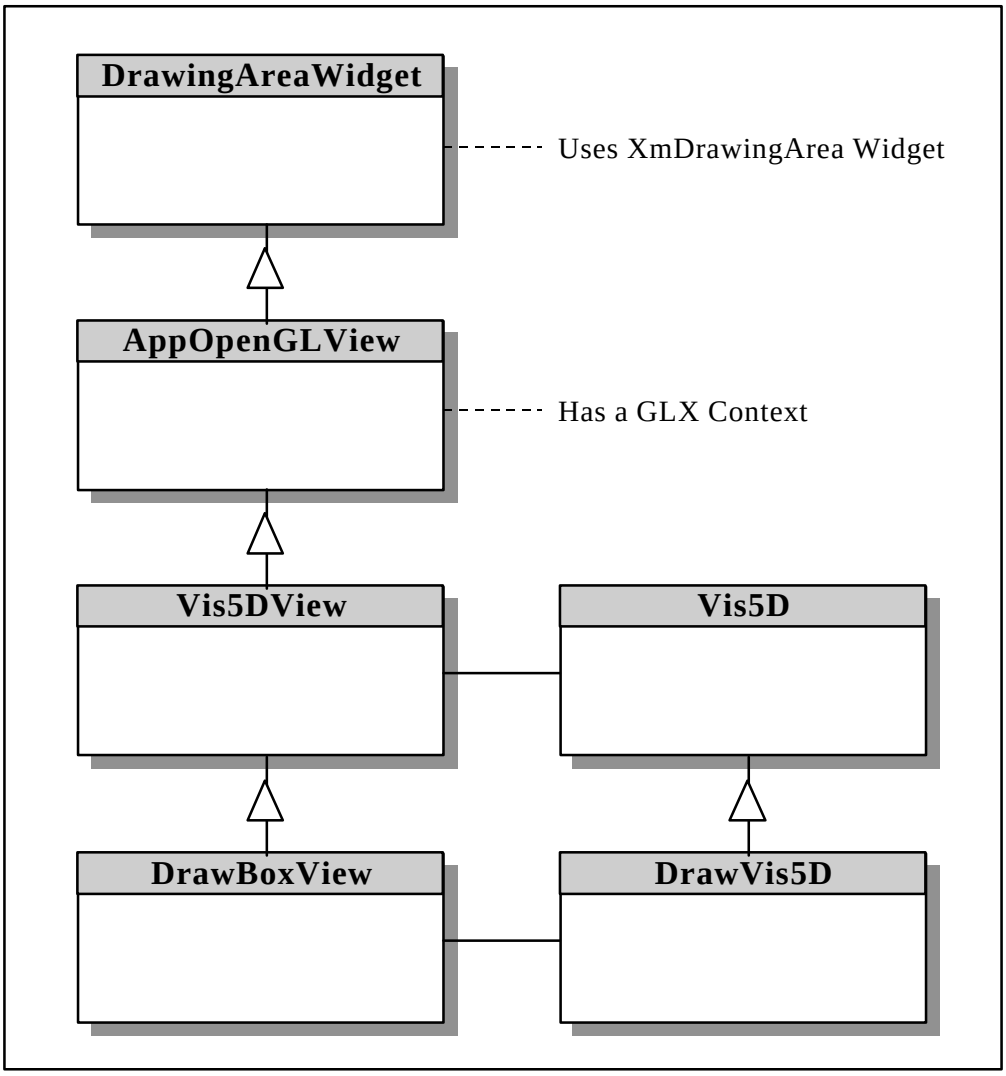


Figure 6-2 TAOS Object Model: libV5DView

Class *DrawingAreaWidget* provides an extensible, polymorphic infrastructure for defining an X/Motif drawing area widget's functionality. A DrawingAreaWidget object references a X/Motif Drawing Area Widget provided by a client to this class. Class *AppOpenGLView* is a specialized DrawingAreaWidget class with a method for configuring the embedding X/Motif application for 3-D OpenGL rendering. The *view* encapsulated by an AppOpenGLView object is double-buffered and therefore capable of supporting animation. Class *Vis5dView* is an AppOpenGLView with mutators responsible for controlling the display (view) of a Vis5D-formatted dataset (file). Class *DrawboxView* extends class Vis5DView to support 2-D rectangle drawing in support of the GriddedWx Editor. Class *Vis5D* collaborates with Vis5DView to provide the embedding application with a full set of services for managing a Vis5D view of large 5D environmental datasets: the client to this library invokes all services provided by the library via the public interface to a Vis5D object (or through the public interface to a derived DrawVis5D object). A Vis5D object manages a single Vis5D OpenGL window; in TAOS Version 1.0 there are no GUI programs that instantiate more than one such window, although there are multiple programs that each have a single Vis5D OpenGL window. Class *DrawVis5D* specializes this class

to support drawing and highlighting of rectangles within the OpenGL model space to support the GriddedWx Editor; public methods are provided to support creation and deletion of rectangle objects in the active OpenGL view.

libUISupport (at: src/OperationsGUI/XMotif/UISupport)

libUISupport is a library of GUI miscellany likely to be made obsolete in a future release. It currently provides a collection of C functions for use by the OperationsGUI in module *UISupport* (files **UISupport.hh** and **UISupport.cc**), and also a C++ class *Visual* that encapsulates methods for converting one or more TAOS Field objects into a Vis5D file (a preprocessing step currently required but likely to be replaced in the future by a direct-memory approach to improve visualization system performance).

6.2 Tcl/Tk Components (at: src/OperationsGUI/TclTk)

The Tcl/Tk GUI components documented in this section include the TAOS Chart Assistant, the Master State Variable Table Editor, and the Integrator and Receiver Configuration interfaces. These applications are each currently implemented as a single Tcl/Tk script. These scripts work in terms of TAOS command extensions to the Tcl scripting language – the C/C++ code that implements the command extensions is located at **src/tcl/tclsh/src**, with source files named with a "tt" prefix as in *ttTable.cc* ("tt" denotes "TAOS Tcl"). The Tcl interpreter which is the target of a **make** for node **src/tcl/tclsh/src** links in the TAOS command extensions but not the Tk extensions required for graphical applications. The Tcl interpreter built as the target of a **make** of **src/tcl/vis5dsh/src** links in the Tk extensions – and a contributed OpenGL widget known as a *togl* widget (Tk OpenGL) – to support TAOS graphical interfaces which may (or may not) include a togl widget for Vis5D-based environmental visualization.

6.2.1 Chart Assistant (at: src/OperationsGUI/TclTk/ChartAssistant)

The Tcl/Tk/togl script **ChartAssistant.tcl**, which runs under the **taos_vis5dsh** Tcl interpreter built from node **src/tcl/vis5dsh**, is an application for creating graphics image files containing weather charts in a format specified by operational military meteorologists (METOC officers) in support of the STOW exercise. This application includes a Vis5D/OpenGL window for environmental data visualization and employs a direct-memory data transfer mechanism to improve performance.

6.2.2 Master State Variable Table Editor (src/OperationsGUI/TclTk/MSVTEditor)

The Tcl/Tk script **MSVTEditor.tcl** implements the Master State Variable Table Editor for managing the contents of the system state variable table stored in file **state-variable.tbl**. This script runs under **taos_vis5dsh**, but does not use a togl widget.

6.2.3 Integrator Configuration Interface (at: src/OperationsGUI/TclTk/IntegConfig)

The Tcl/Tk script **IntegConfig.tcl** implements the configuration interface for the TAOS Environmental Integrator. This program is launched from the *Integrator* option menu (action *Configure*) on the main interface. This script runs under **taos_vis5dsh**, but does not use a togl widget.

6.2.4 Receiver Configuration Interface (at: src/OperationsGUI/TclTk/RcvrConfig)

The Tcl/Tk script **RcvrConfig.tcl** implements the configuration interface for all TAOS Receivers and is invoked when the user pressed a Receiver button on the main interface or the *Configure* button for a Receiver from the Integrator Configuration interface. This script runs under **taos_vis5dsh** but does not use a togl widget.